Q1：new和delete是什么？

#include<iostream>

using namespace std;

int main()

{

    使用 new 分配内存

1. 为单个变量分配内存

Type\* pointer = new Type;

解释：

new Type：在堆上分配一个 Type 类型的对象，并返回该对象的地址。

pointer 是一个指针，指向分配的内存空间。

int\* p = new int; // 在堆上分配一个 int 类型的内存

\*p = 42;          // 给分配的内存赋值

std::cout << \*p << std::endl; // 输出：42

2. 为数组分配内存

Type\* pointer = new Type[size];

解释：

new Type[size]：在堆上分配一个包含 size 个 Type 类型元素的数组，并返回数组的第一个元素的地址。

示例：

int\* arr = new int[5]; // 分配一个包含 5 个 int 的数组

for (int i = 0; i < 5; ++i) {

    arr[i] = i \* 10;  // 给数组元素赋值

}

for (int i = 0; i < 5; ++i) {

    std::cout << arr[i] << " "; // 输出：0 10 20 30 40

}

std::cout << std::endl;

使用 delete 释放内存

动态分配的内存不会被自动回收，你必须手动释放它，否则会出现内存泄漏。

1. 释放单个变量

delete pointer;

示例：

int\* p = new int; // 分配内存

\*p = 42;          // 使用内存

delete p;         // 释放内存

p = nullptr;      // 避免悬空指针

2. 释放数组

delete[] pointer;

示例：

cpp

复制

int\* arr = new int[5]; // 分配数组

// 使用数组

delete[] arr;          // 释放数组内存

arr = nullptr;         // 避免悬空指针

注意事项

1. 不要重复释放内存

释放同一块内存两次会导致未定义行为：

cpp

复制

int\* p = new int;

delete p;

delete p; // 错误！p 已经被释放

2. 避免悬空指针

释放内存后，指针仍然保存着已释放内存的地址。最好将指针设置为 nullptr：

cpp

复制

int\* p = new int;

delete p;

p = nullptr; // 避免悬空指针

3. new 和 delete 必须匹配

使用 new 分配单个变量，必须用 delete 释放。

使用 new[] 分配数组，必须用 delete[] 释放。

错误示例：

int\* p = new int[5];

delete p; // 错误！应该使用 delete[]

完整示例代码

#include <iostream>

int main() {

    // 动态分配单个变量

    int\* singleVar = new int; // 分配内存

    \*singleVar = 100;         // 使用内存

    std::cout << "Single variable: " << \*singleVar << std::endl;

    delete singleVar;         // 释放内存

    singleVar = nullptr;      // 避免悬空指针

    // 动态分配数组

    int size = 5;

    int\* array = new int[size]; // 分配数组

    for (int i = 0; i < size; ++i) {

        array[i] = i + 1;       // 使用数组

    }

    std::cout << "Array elements: ";

    for (int i = 0; i < size; ++i) {

        std::cout << array[i] << " "; // 输出数组元素

    }

    std::cout << std::endl;

    delete[] array;            // 释放数组内存

    array = nullptr;           // 避免悬空指针

    return 0;

}

运行结果：

Single variable: 100

Array elements: 1 2 3 4 5

总结

使用 new 动态分配内存，使用 delete 手动释放内存。

区分单个变量和数组的分配与释放：new 对应 delete，new[] 对应 delete[]。

小心内存泄漏和悬空指针。

    return 0;

}

Q2：如何进行运算符重载

Q3：this->是什么

在 C++ 中，this 是一个指针，指向当前对象自身。当你在类的成员函数中使用 this 时，它指向调用该函数的对象，允许你在类的成员函数中访问当前对象的成员变量和成员函数。

this-> 是一种常见的用法，用于明确表示当前对象的成员（变量或函数）。虽然在大多数情况下可以直接使用成员的名称（例如 member），但使用 this->member 可以提高代码的可读性，尤其是在以下情况下：

**1. 解决成员变量和局部变量的命名冲突**

如果成员变量和函数的局部变量（或参数）同名，this-> 可以区分它们。例如：

#include <iostream>

using namespace std;

class MyClass {

private:

int value;

public:

MyClass(int value) {

// 使用 this-> 明确指定是成员变量，而不是参数

this->value = value;

}

void printValue() {

cout << "Value: " << this->value << endl;

}

};

int main() {

MyClass obj(42);

obj.printValue(); // 输出: Value: 42

return 0;

}

在上面的代码中，构造函数的参数和成员变量都叫 value。为了避免混淆，使用 this->value 表示成员变量，而单独的 value 表示参数。

**2. 在模板或继承中消除歧义**

在某些情况下，例如模板类或继承结构中，使用 this-> 明确表示一个成员是来自当前类的，而不是全局范围或基类中的成员。例如：

#include <iostream>

using namespace std;

class Base {

protected:

int x = 10;

};

class Derived : public Base {

public:

void print() {

// 使用 this-> 明确表示 x 是来自基类的成员

cout << "x = " << this->x << endl;

}

};

int main() {

Derived obj;

obj.print(); // 输出: x = 10

return 0;

}

在复杂的模板代码中，使用 this-> 可以帮助编译器正确解析成员变量或函数。

**3. 在链式调用中使用 this**

this 指针还可以用于返回当前对象的引用，从而实现链式调用。例如：

#include <iostream>

using namespace std;

class MyClass {

private:

int value;

public:

MyClass& setValue(int value) {

this->value = value; // 使用 this-> 明确表示成员变量

return \*this; // 返回当前对象的引用

}

void printValue() {

cout << "Value: " << this->value << endl;

}

};

int main() {

MyClass obj;

obj.setValue(42).printValue(); // 链式调用

return 0;

}

这里的 this->value = value 和 return \*this 是链式调用的关键。

**4. 提高代码可读性**

即使没有命名冲突或继承关系，有些开发者仍然喜欢使用 this-> 来明确表示某个变量是当前对象的成员变量。这种风格在大型代码库中可能会更受欢迎，因为它让代码更清晰。

**总结**

* this 是一个指针，指向当前对象。
* this-> 常用于：
  1. 解决成员变量和局部变量的命名冲突。
  2. 在继承或模板代码中消除歧义。
  3. 实现链式调用。
  4. 提高代码可读性。

你可以根据实际需求决定是否使用 this->，但要记住它的主要作用是让代码更清晰，避免歧义。

Q4:运算符重载